

Figure 1

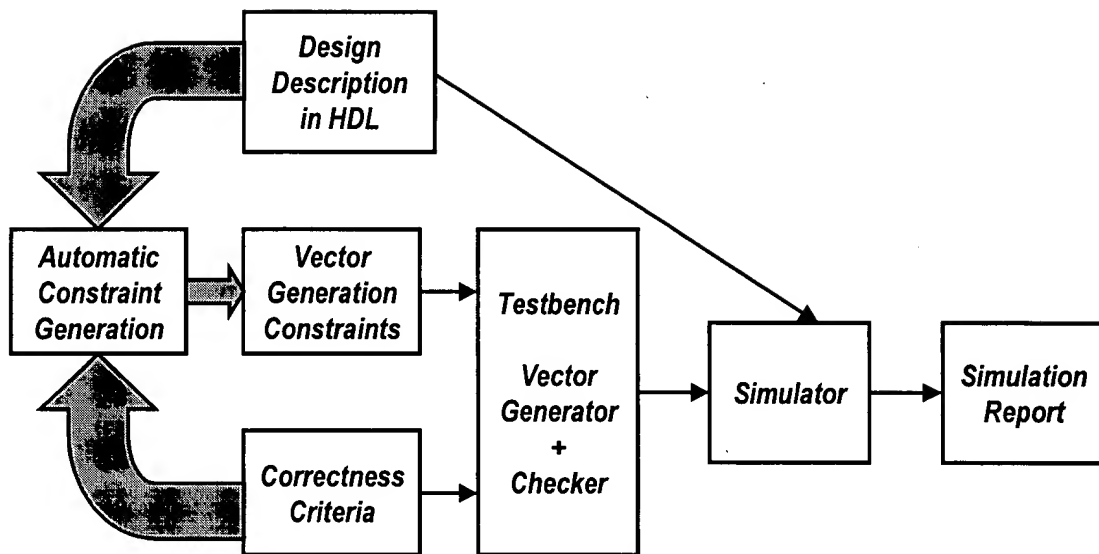


Figure 2

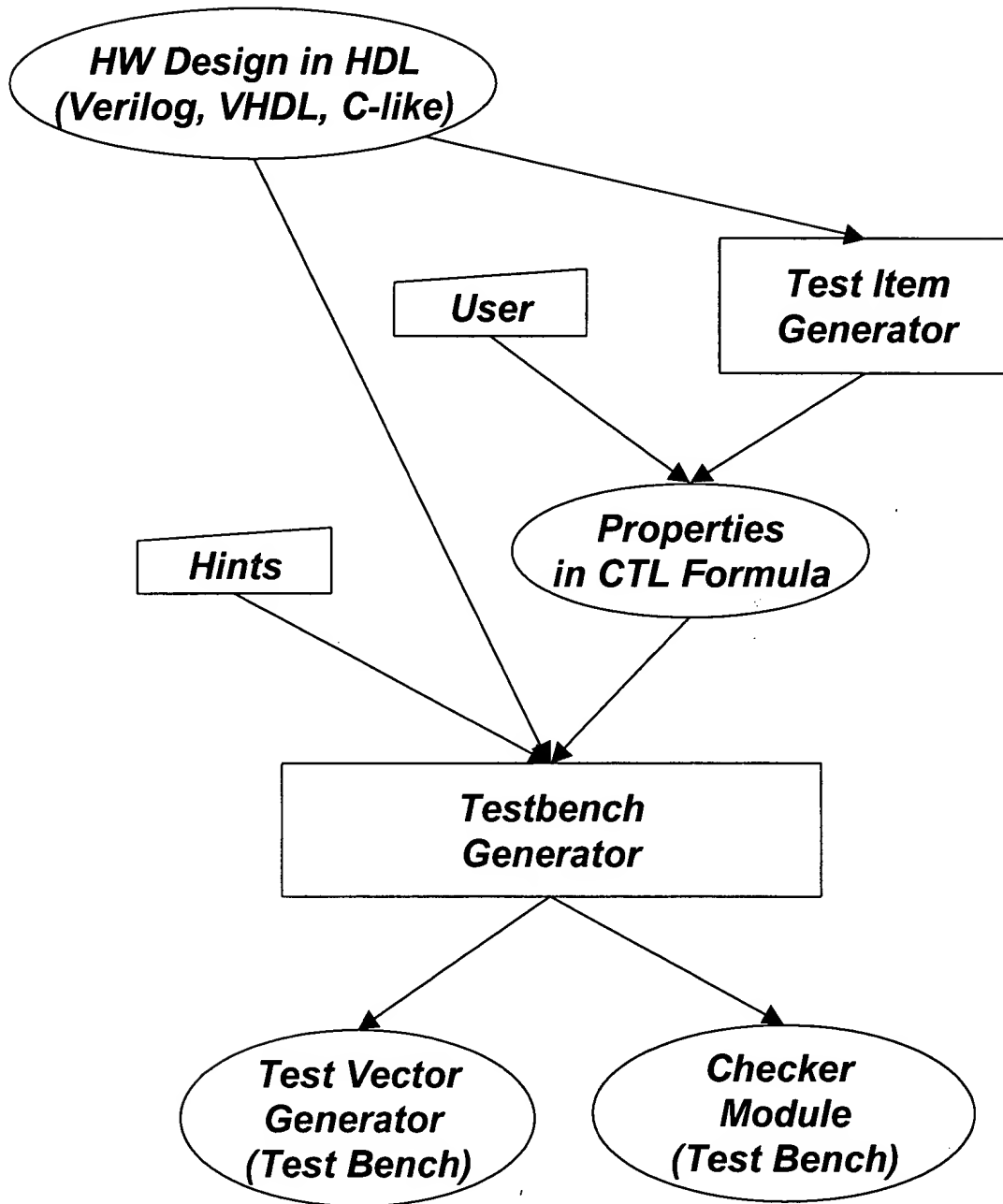


Figure 3

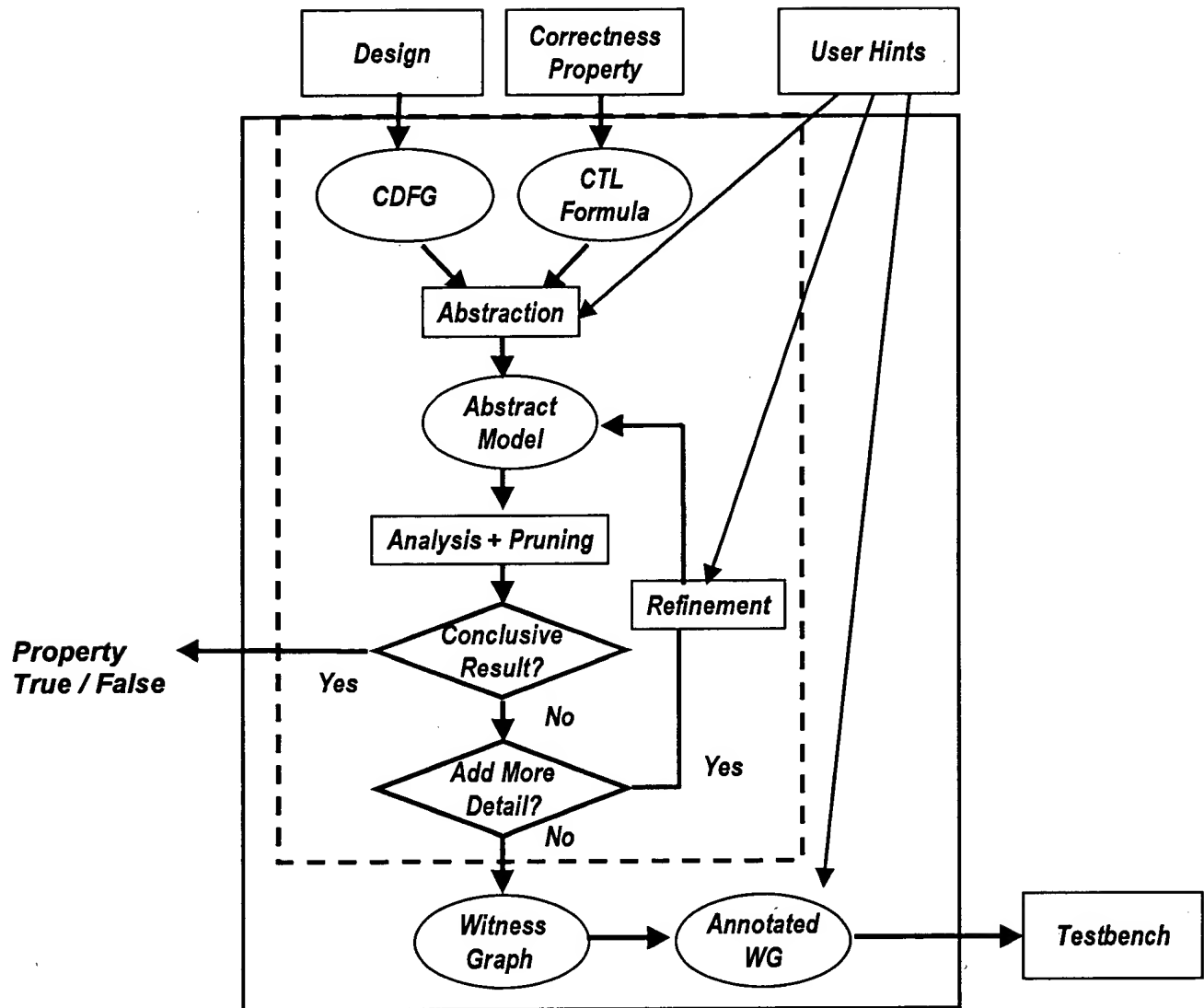


Figure 4

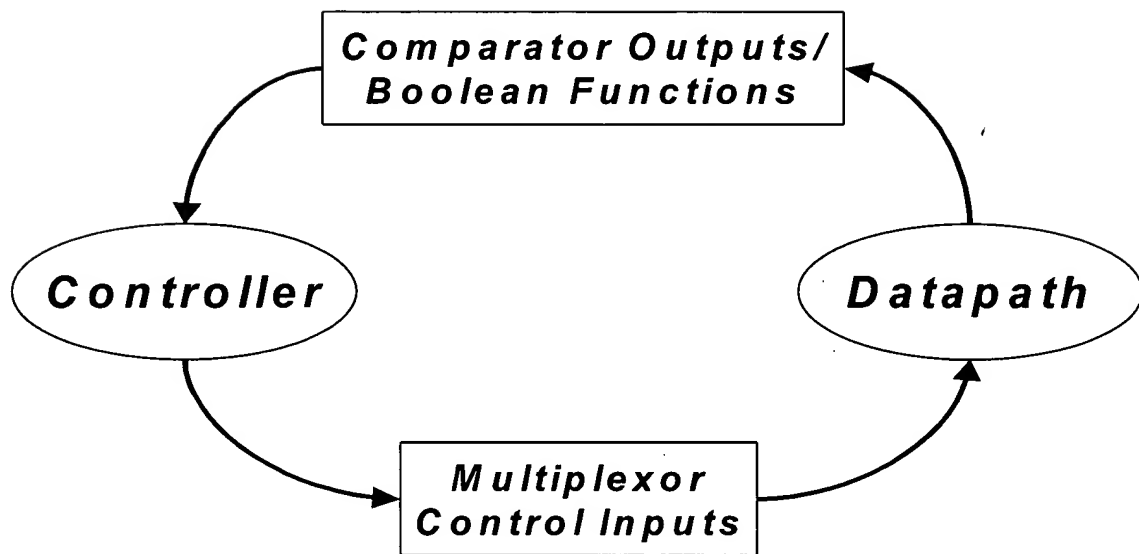


Figure 5

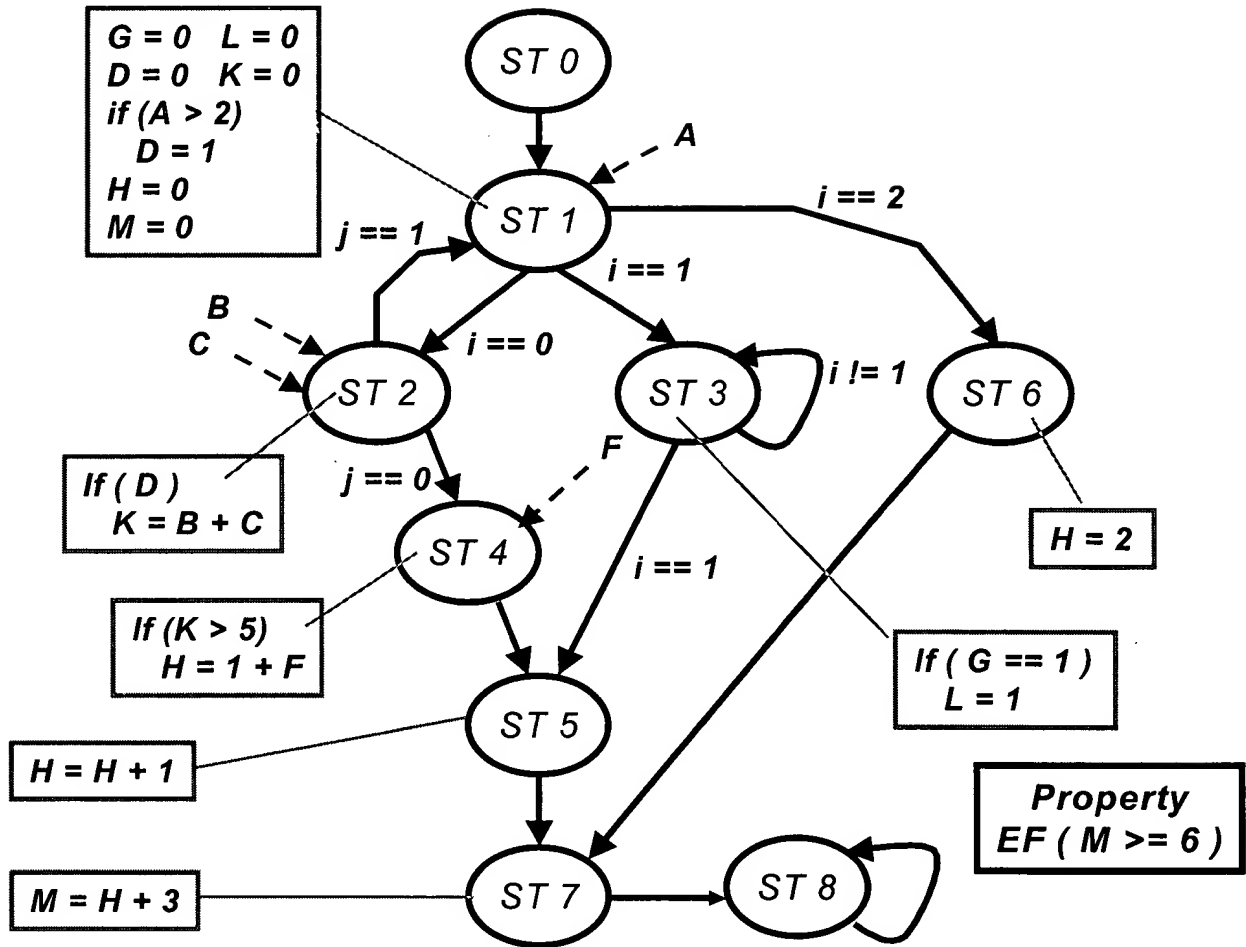


Figure 6

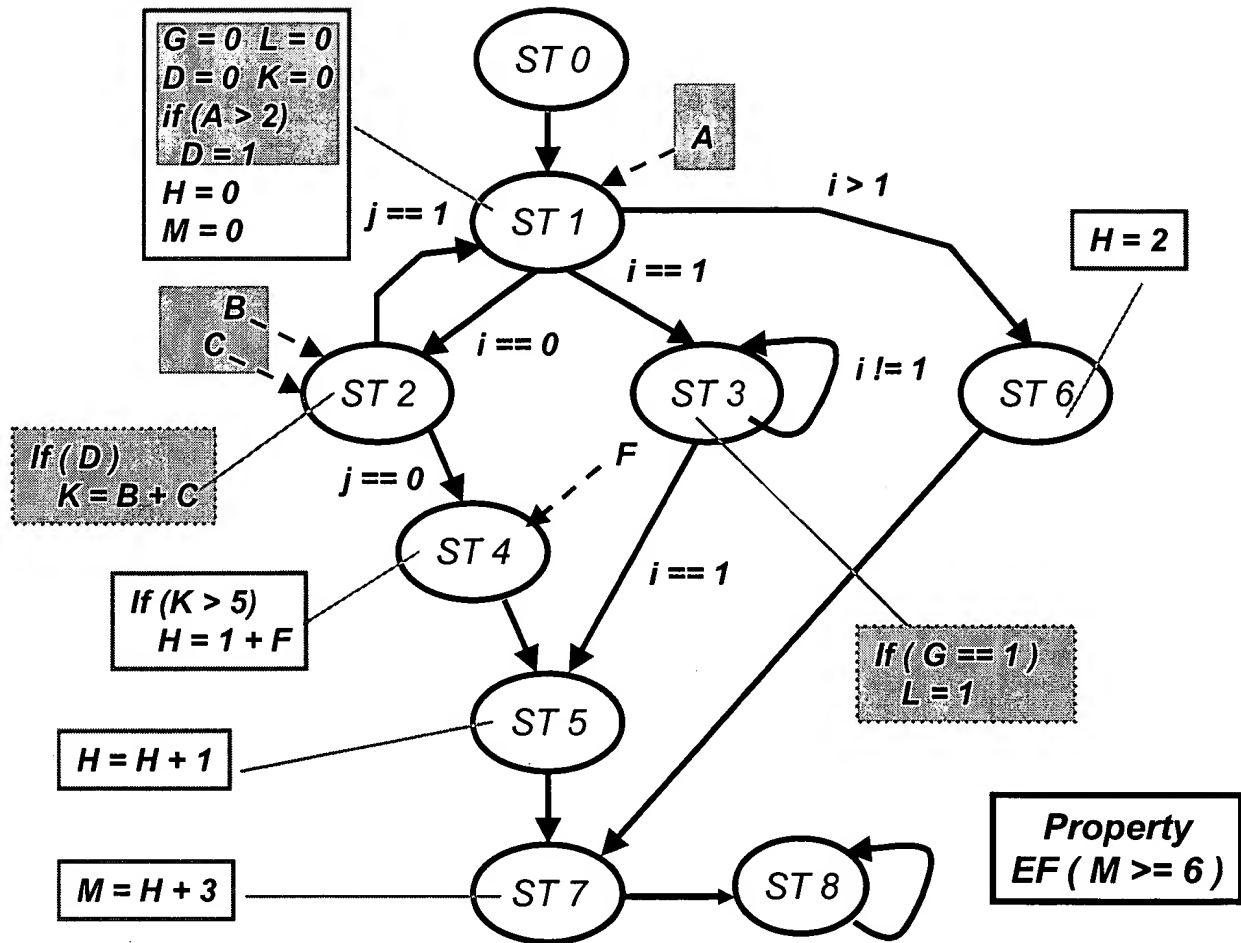


Figure 7

8 / 21

```

mc_for_sim(model m, ctlFormula f) {
    ctlFormula f1, f2, negf;
    states upper, upper1, upper2=NULL, negative=NULL;
    // handle subformulas recursively

    if (f1 = leftChild(f)) {
        mc_for_sim(m, f1);
        upper1 = get_upper(f1);
    }

    if (f2 = rightChild(f)) {
        mc_for_sim(m, f2);
        upper2 = get_upper(f2);
    }

    // case analysis on operator at this level
    switch(type(f)) {
        case TRUE: upper = ALL; break;
        case FALSE: upper = NULL; break;
        case ATOMIC: upper = mc_atomic(m, f); break;
        case NOT: upper = complement(upper1); break;
        case AND: upper = and(upper1, upper2); break;
        case OR: upper = or(upper1, upper2); break;
        case EX: case EF: case EU: case EG:
            upper = mc_etype(upper1, upper2); break;
        default: // A-type operators left
            switch(type(f)) {
                case AX: upper = mc_ex(upper1); break;
                case AF: upper = mc_ef(upper1); break;
                case AU: upper = mc_eu(upper1, upper2); break;
                case AG: upper = mc_eg(upper1); break;
            }
    }

    // compute negative sets also
    negf = negate(f);
    mc_for_sim(m, negf);
    negative = and(upper, get_upper(negf)); break;
}

// associate the sets with f
associate(f, upper, negative);
}

```

Figure 8


```
check_mc (model m, ctlFormula f)
{
    mc_for_sim(m, f);
    if (initState(m) ∉ get_upper(f))
        result = PROPERTY_FALSE;
    else if (A-type(f) &&
        initState(m) ∉ get_negative(f))
        result = PROPERTY_TRUE;
    else
        result = INCONCLUSIVE;
    return result;
}
```

Figure 9

```

mark_witness_top(model m, ctlFormula f)
{
    reachable = compute_reachable(m, initState(m));

    switch(type(f)){
        case AX: case AF: case AU: case AG:
            witness_top= and(get_negative(f), reachable);
            break;
        default:
            witness_top= and(get_upper(f), reachable);
    }

    mark_states(witness_top);
    mark_witness_rec(m, f, witness_top);
}

```

Figure 10a

```

mark_witness_rec(model m, ctlFormula f, states careSet)
{
    states witness, negWitness, subWitness;
    // associate witness set for f
    witness = and(get_upper(f), careSet);
    associate_witness(f, witness);
    // recursive calls with modified careSets
    switch(type(f)) {

        case TRUE: case FALSE: case ATOMIC: case NOT:
            break;

        case AND: case OR:
        case EF: case EU: case EG:
            mark_witness_rec(m, leftChild(f), witness);
            if (rightChild(f) != NULL)
                mark_witness_rec(m, rightChild(f), witness);
            break;

        case EX:
            subWitness = compute_image(m, witness);
            // mark additional states
            mark_states(subWitness);
            mark_witness_rec(m, leftChild(f), subWitness);
            break;

        case AX: case AF: case AU: case AG:
            negWitness = and(get_negative(f), careSet);
            associate_neg_witness(f, negWitness);
            mark_witness_rec(m, negate(f), negWitness);
            break;

    }
}

```

Figure 10b

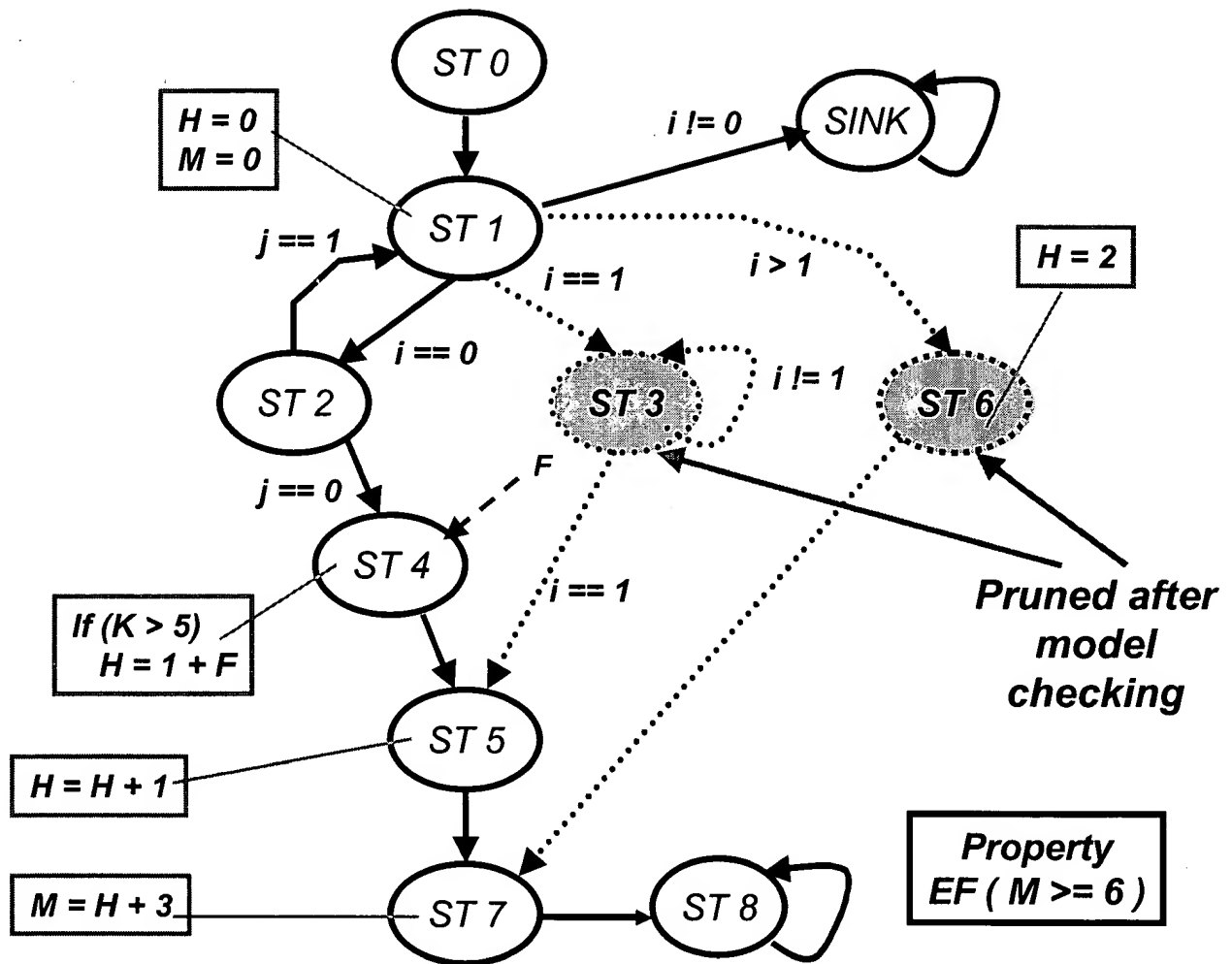


Figure 11

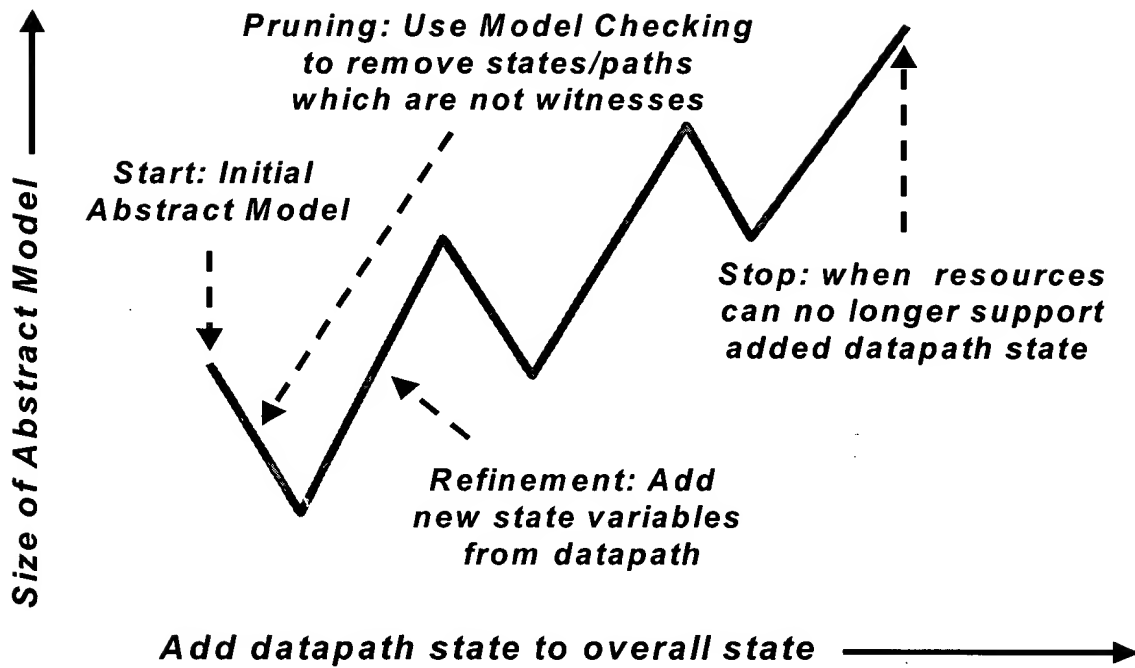


Figure 12

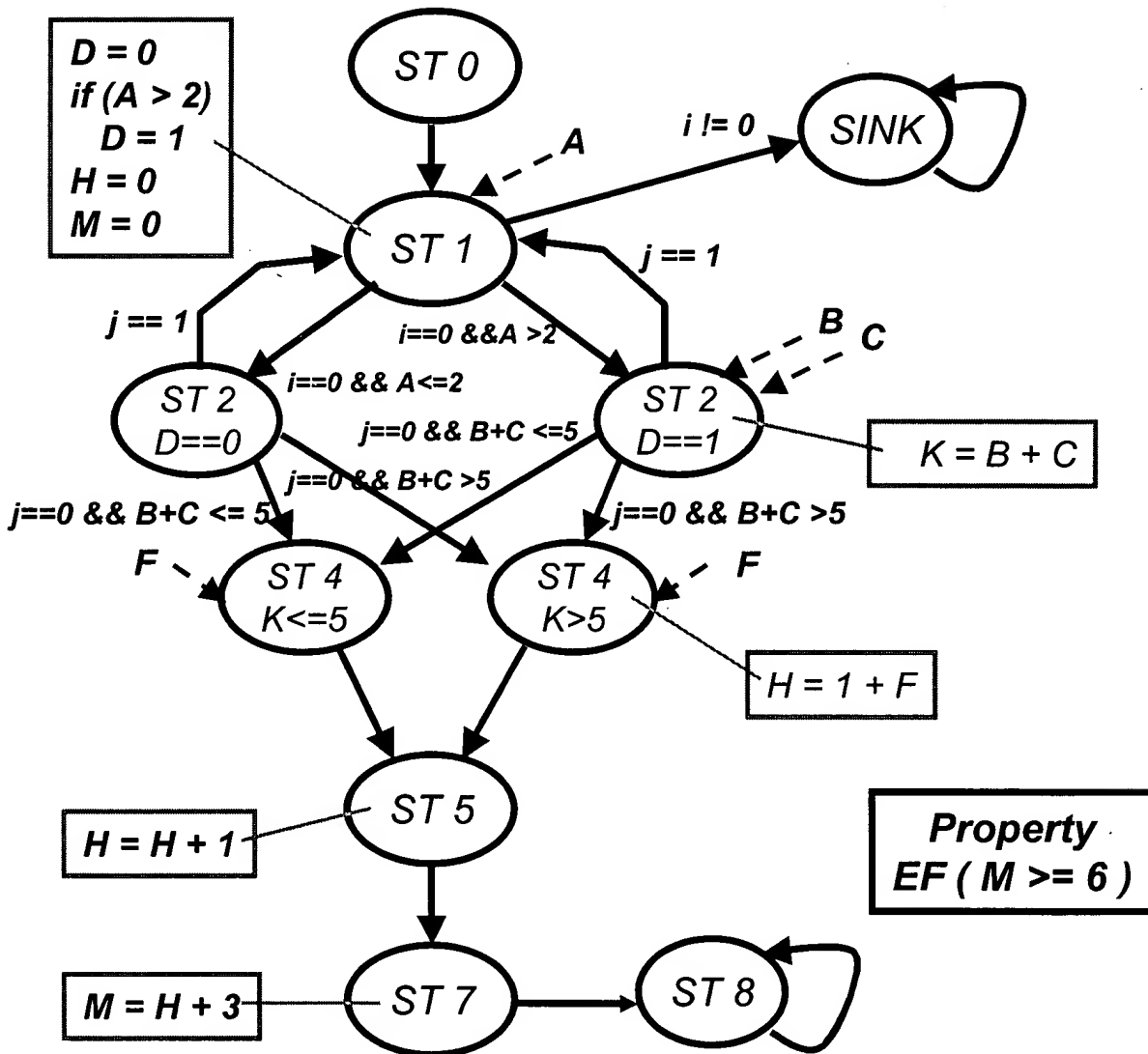


Figure 13

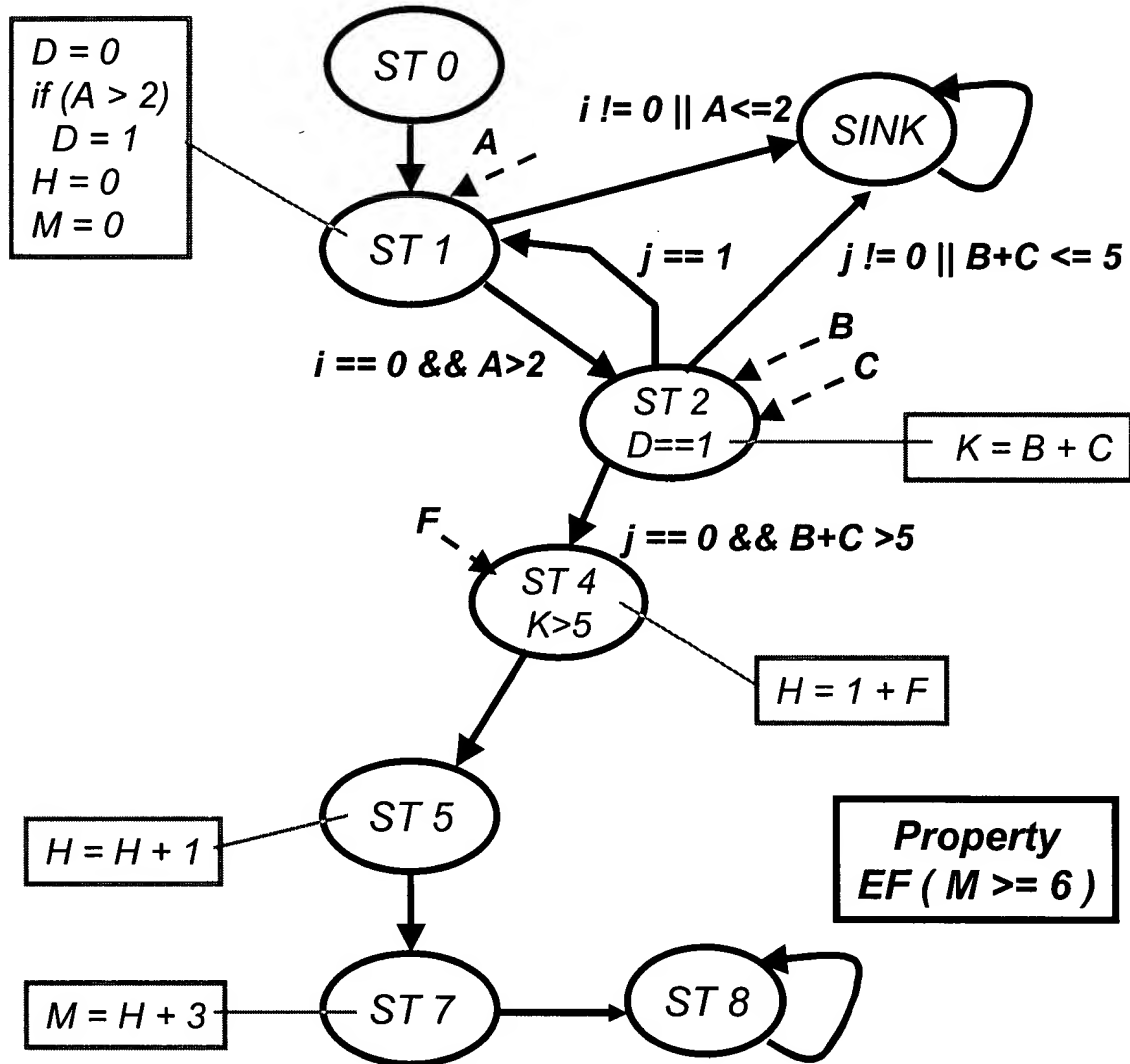


Figure 14

```

witness_sim(design d, ctlFormula f, state s)
{
    states w, w1;
    int result, neg_result;
    w = get_witness(f);
    w1 = get_witness(leftChild(f));
    // case analysis on operator at this level

    switch(type(f)) {

        case TRUE: result = SUCCESS; break;

        case FALSE: result = FAILURE; break;

        case ATOMIC: result = satisfies(s,f); break;

        case NOT: result=satisfies(s,negate(f));break;

        case AND:
            result = witness_sim(d,leftChild(f),s);
            if (result==SUCCESS)
                result = witness_sim(d,rightChild(f),s);
            break;

        case OR:
            result = witness_sim(d,leftChild(f),s);
            if (result==FAILURE)
                result = witness_sim(d,rightChild(f),s);
            break;

        case EX:
            foreach state t, abs(t) ∈ w1, {
                if (exists_transition(s,t)){
                    result = witness_sim(d,leftChild(f),t);
                    if (result==SUCCESS) break;
                }
            }
            break;
    }
}

```

Figure 15a


```

case EF:
    foreach state t, abs(t) ∈ w1, {
        if (path = find_a_path(s,t)){
            result = witness_sim(d, leftChild(f), t);
            if (result == SUCCESS) break;
        }
    }
    break;

case EU:
    result = witness_sim(d, rightChild(f), s);
    if (result == FAILURE) {
        mark(s, f);
        result = witness_sim(d, leftChild(f), s);
        if (result == SUCCESS)
            foreach unmarked state t, abs(t) ∈ w {
                if (exists_transition(s,t)){
                    result = witness_sim(d, f, t);
                    if (result == SUCCESS) break;
                }
            }
    }
    break;

case EG:
    result = witness_sim(d, leftChild(f), s);
    if (result == SUCCESS) {
        mark(s, f);
        if (exists_transition_to_marked(s, f))
            result = SUCCESS;
    }
    else
        foreach unmarked state t, abs(t) ∈ w {
            if (exists_transition(s,t)){
                result = witness_sim(d, f, t);
                if (result == SUCCESS) break;
            }
        }
    }
    break;

```

Figure 15b

```

case AX: case AF: case AU: case AG:
    if (abs(s) ≠ get_neg_witness(f))
        result = SUCCESS;
    else {
        // generate counter-example for !f
        neg_result = witness_sim(d,negate(f),s);
        result = (neg_result == SUCCESS) ?
            FAILURE : SUCCESS;
    }
} // end switch
return result;

```

Figure 15c

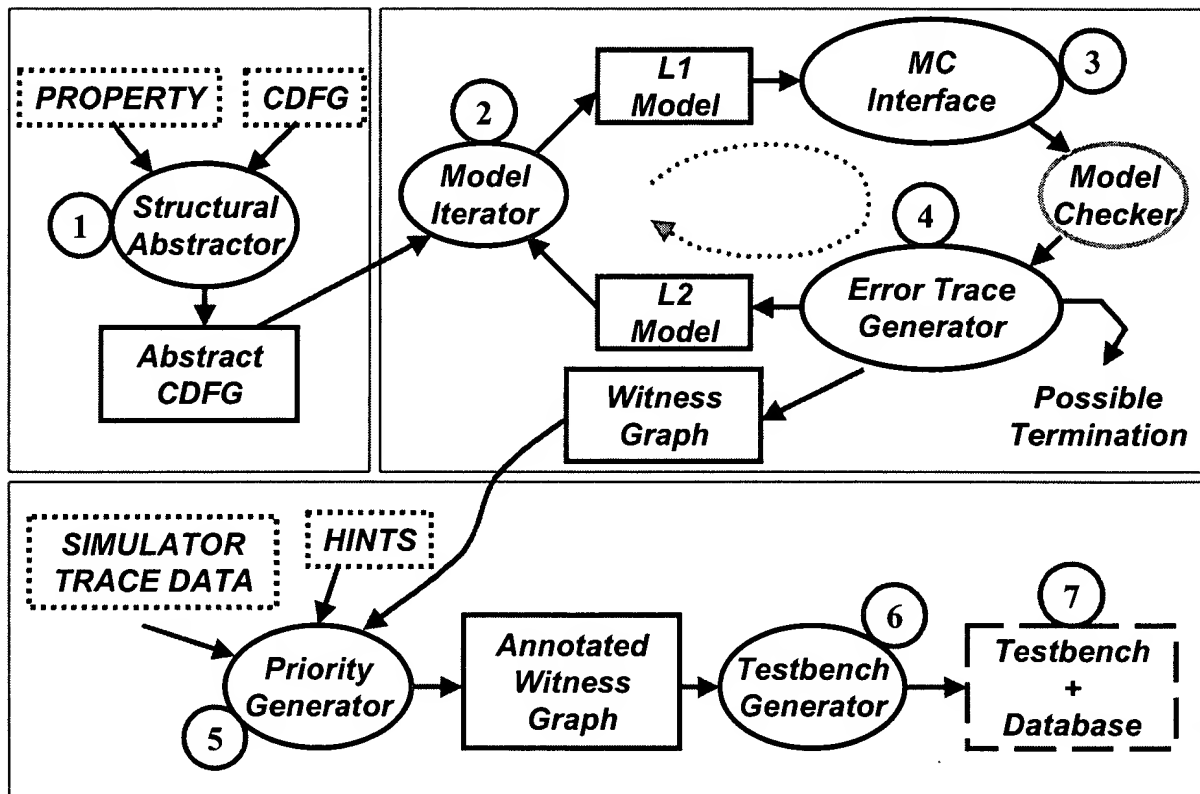


Figure 16

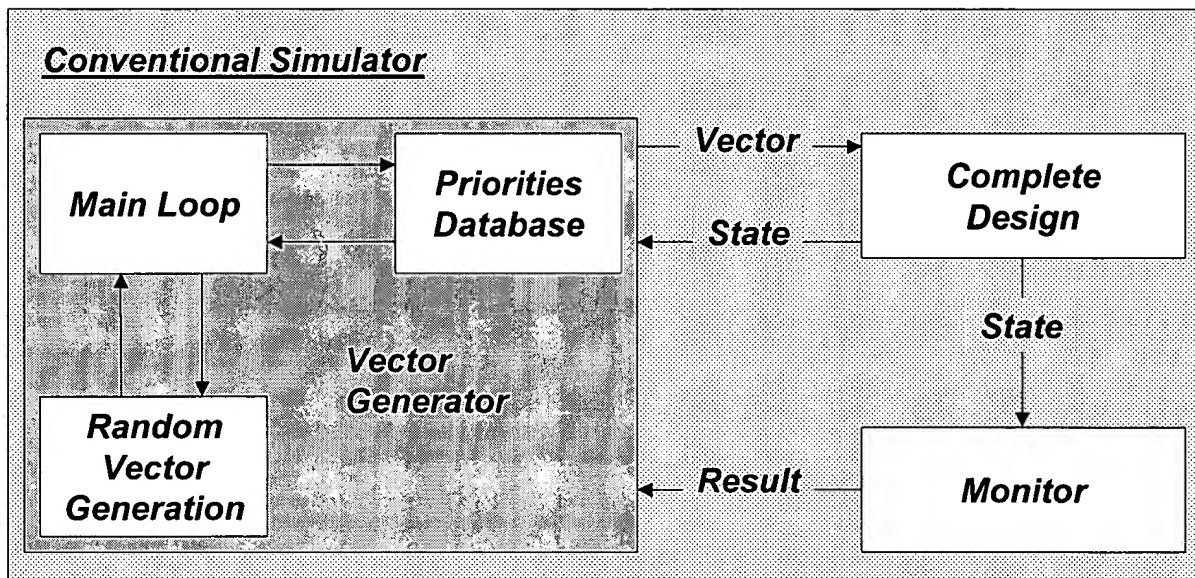


Figure 17

```

Testbench() {
    ...
    do {
        determine current state of design;
        determine abstract state from current state;
        query database for desirable transition;
        if (input vector NOT in database) {
L1: input vector = random vector;
            if (input vector satisfies condition) {
                simulate input vector;
                if (next abstract state != desired) {
                    roll back simulation one cycle;
                    go to L1;
                }
            }
        }
    } while (property is not yet proved/disproved);
    ...
}

```

Figure 18